TECHNOLOGY GUIDE

Intel Corporation

Telemetry Aware Scheduling (TAS) - Automated Workload Optimization with Kubernetes (K8s*) Technology Guide

Authors

Dave Cremins

Killian Muldoon

Swati Sehgal

1 Introduction

Kubernetes* (K8s*) is becoming the de-facto standard for managing multiple application deployment models across a diverse range of workloads from Internet of Things (IoT) and Artificial Intelligence (AI) to Network Function Virtualization (NFV). To serve these use cases, the K8s control plane needs to evolve to meet stringent networking and resource management requirements for optimum utilization of compute, network, and other resources. Automated, data-driven actions are an indispensable tool in the orchestration layer to achieve this, generating reduced expenditure (both capital and operational), greater operational efficiency, improved service performance, and a more robust customer experience overall.

This paper presents a methodology for enhancing scheduling capabilities in K8s, empowering the system to make more informed placement decisions. Telemetry Aware Scheduling (TAS) enables automated actions and intelligent placement of workloads based on up-to-date platform telemetry.

This technology guide's audience includes cluster operators, solutions architects, and developers looking to introduce data driven pod placement to improve utilization, efficiency and effectiveness across their clusters. Readers will learn about the existing gaps in K8s scheduling capabilities and how TAS overcomes these problems. This document covers the deployment process and usage of TAS to manage initial workload placement and workload distribution through the lifecycle.

This technology guide is part of the Network Transformation Experience Kit, which is available at https://networkbuilders.intel.com/network-technologies/network-transformation-exp-kits.

Table of Contents

Figures

Tables

1.1 Terminology

Table 1. Terminology

ABBREVIATION	DESCRIPTION
AI	Artificial Intelligence
API	Application Programming Interface
CPU	Central Processing Unit
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
Intel® RDT	Intel® Resource Director Technology
IoT	Internet of Things
JSON*	JavaScript* Object Notation
K8s*	Kubernetes*
LLC	Last Level Cache
NFV	Network Function Virtualization
OPNFV	Open Platform for NFV
PMU	Performance Monitoring Unit
RAM	Random Access Memory
RAS	Reliability, Availability, and Serviceability
TAS	Telemetry Aware Scheduling – a Kubernetes scheduler add on
VPU	Vision Processing Units

1.2 Reference Documentation

Table 2. Reference Documents

REFERENCE	SOURCE
Barometer Home	https://wiki.opnfv.org/display/fastpath/Barometer+Home
collectd	https://github.com/collectd/collectd
Closed Loop Automation - Telemetry Aware Scheduler for Service Healing and Platform Resilience Demo	https://networkbuilders.intel.com/closed-loop-automation-telemetry-aware-scheduler-for-service-healing-and-platform-resilience-demo
Closed Loop Automation - Telemetry Aware Scheduler for Service Healing and Platform Resilience White Paper	https://builders.intel.com/docs/networkbuilders/closed-loop-platform-automation-service-healing-and-platform-resilience.pdf
Energy Savings & Resiliency with Closed Loop Platform Automation	https://tma.roc.cnam.fr/Proceedings/TMA_Demo_5.pdf
Intel_RDT	https://wiki.opnfv.org/display/fastpath/Intel_RDT
Kubernetes Descheduler User Guide	https://github.com/kubernetes-sigs/descheduler/blob/master/docs/user-guide.md
K8s Prometheus* Adapter	https://github.com/DirectXMan12/k8s-prometheus-adapter
Node Exporter	https://github.com/prometheus/node_exporter
One Click Install of Barometer Containers	https://wiki.opnfv.org/display/fastpath/One+Click+Install+of+Barometer+Containers
Prometheus	https://github.com/prometheus/prometheus
Telemetry Aware Scheduling	https://github.com/intel/telemetry-aware-scheduling

2 Overview

TAS is a K8s* add-on that consumes platform metrics and makes intelligent scheduling decisions based on operator-defined policies. TAS can be used to direct workloads to specific nodes based on up-to-date platform telemetry during pre-runtime. For example, when the pod is initially started on the cluster and during runtime, the workload is already up and running on the platform.

TAS implements a K8s scheduler extender, meaning it modifies the decisions made by the core K8s system, rather than replacing them entirely. The extender software is open source and available at: https://github.com/intel/telemetry-aware-scheduling. The standard resource checking, workload awareness, and other scheduling rules remain unchanged, and all standard pods are compatible with a cluster running TAS. Telemetry is collected using the K8s custom metrics Application Programming Interface (API), a standard metric interface in the orchestrator.

Placement decisions are modified by TAS in three ways:

- Filtering
- Prioritization
- Descheduling

Each one can be customized as an individual telemetry scheduling policy, and all three can be combined into a single telemetry scheduling policy for holistic workload placement automation.

A telemetry scheduling policy, which is defined as a K8s custom resource, contains a set of rules like those described below. Each policy may serve more than one type of workload, and more than one policy can be active in a cluster at any time. Each scheduling decision made by TAS is based on a single policy.

Filtering imposes telemetry-driven rules and removes nodes that break those rules from the list of candidates. An example of filtering rule is:

- Don't schedule on nodes if the power usage is greater than 90 percent capacity.

Figure 1 shows an example of Kubernetes scheduler filtering based on the rule using a three-node cluster, with Node C ruled out as a candidate because of its high power use.

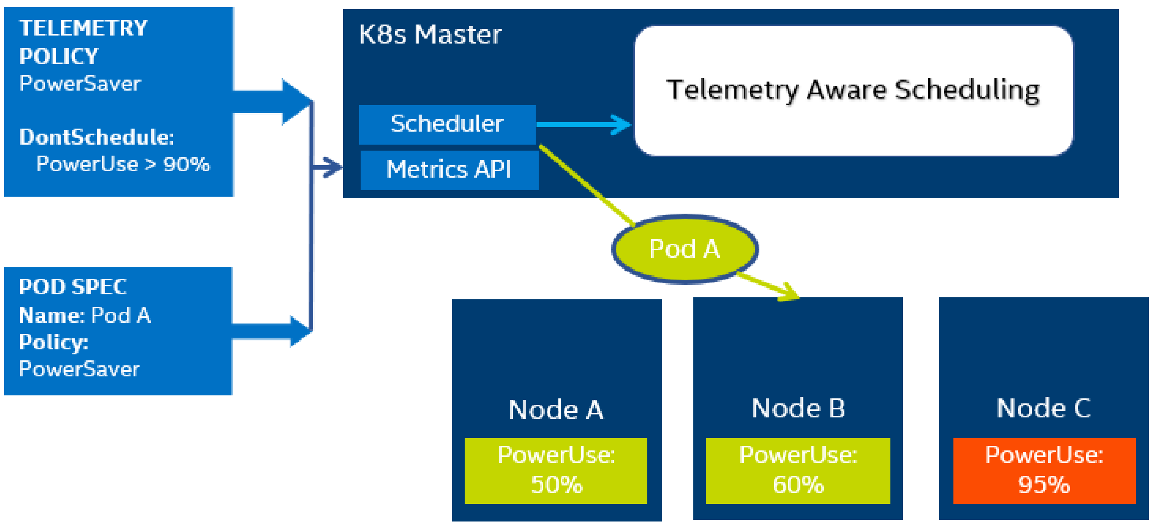

Figure 1. TAS Filter in Action

Prioritization ranks candidate nodes in order of suitableness based on a telemetry rule. A prioritization rule example is:

- Schedule on node with highest free memory

Figure 2 shows Pod A being directed to Node A because it has the highest amount of RAM available of all the nodes.

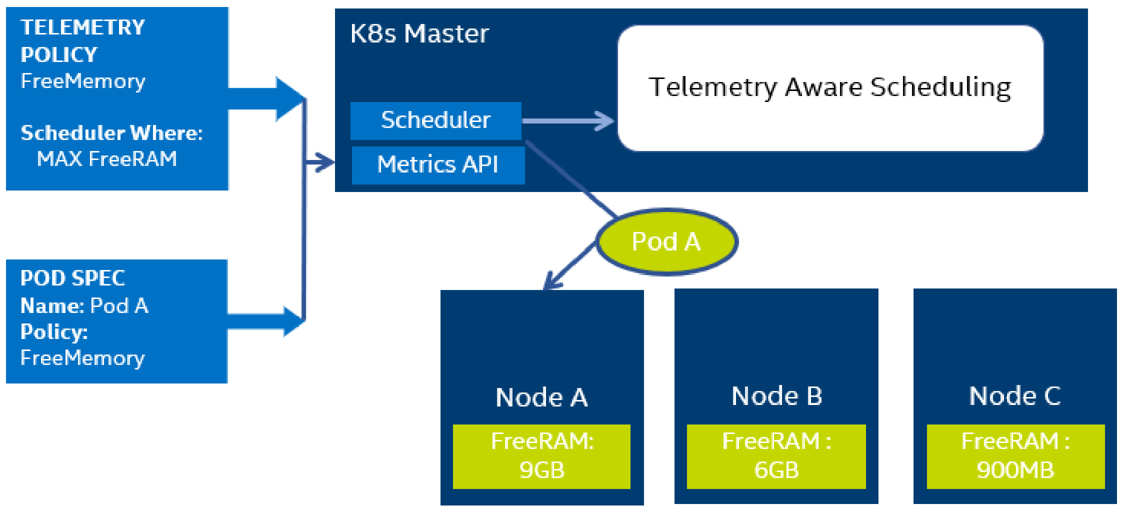

Figure 2. Prioritize in Action

Descheduling actively removes a workload from its host, causing it to be placed on a more suitable host. A descheduling rule example is: deschedule the workload if the temperature is greater than 90°.

In Figure 3, there is a pod already running on Node A, however, its deschedule policy, part of the "Temperature" Telemetry Policy, is broken. Descheduler, which inspects the cluster for policy breakers at regular intervals, will deschedule the pod. The workload will subsequently start up on a different node.

Note: The service provided by the workload is not available during the period from the deschedule to the application providing service on the next scheduled platform. High Availability and resiliency schemes can make use of this type of activity as one part of a set of overall system level resiliency solutions. For some examples shown, metrics are provided from the platform by the collectd* telemetry agent (https://github.com/collectd/collectd) and plugins available in the Open Platform for NFV (OPNFV) Barometer project (https://wiki.opnfv.org/pages/viewpage.action?pageId=16220285).

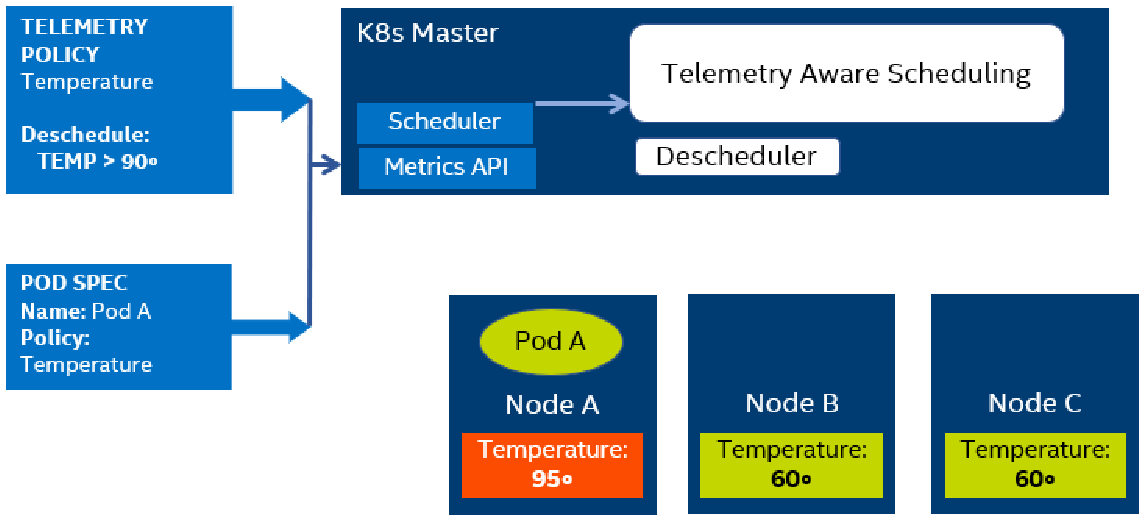

Figure 3. Deschedule in Action

2.1 Challenges Addressed

In K8s, scheduling is the selection of a suitable worker node to run a specific workload and it is focused on discrete pre-defined resources. A node is defined as suitable if certain resources are not explicitly claimed, rather than the current or historical level of utilization.

First class resources, CPU, and memory can be claimed by specifying their expected usage values before deploying a workload, but the scheduler tracks these static claims rather than up-to-date usage. This can lead to a situation where a node with very little CPU usage is ignored by the K8s scheduler, and the K8s scheduler directs new workloads to nodes with higher CPU contention.

For other resources, for example GPUs, Field-Programmable Gate Arrays (FPGAs), network cards and more, there is no way to link workload placement to utilization of these resources. They are made available only as atomic devices, with a unit either labelled as **in use** or **free**.

K8s's blindness to utilization and platform telemetry creates an inability to respond correctly to the current state of the cluster. Manifestations of this failure include:

- Unbalanced pod placement
- Unexpected resource utilization
- Ignorance of red flags that could be used to predict node or device faults

2.2 Use Cases

TAS can enforce rules and automatically manage resource utilization across a cluster. The system architecture is generic and can be used to help solve any scheduling problem where platform telemetry is a useful input. Example use cases include:

- noisy neighbors
- Vision Processing Unit (VPU) allocation
- power management and platform resilience automization.

Discussions of each case are described in the following sub-sections.

2.2.1 Noisy Neighbor Protection

Workloads on the same host, also known as co-located workloads, can compete for specialized resources in ways invisible to K8s. A classic example for this "noisy neighbor" problem occurs through overuse of the Last Level Cache (LLC) on a machine. By causing too many LLC misses, one workload starves another of access to the cache. The performance impact can be significant for certain workloads.

The result can be issues that are hard to trace, due to complex performance affecting interactions between workloads. The K8s default scheduler provides no way to track specific types of resource usage like LLC thrashing.

TAS can detect these issues and direct performance-sensitive workloads away from compromised nodes. A rule set like the following example set can counteract the issues – turning workloads away from nodes on which they're likely to perform poorly.

- Don't schedule if cache hit ratio is less than 79
- Schedule on node with greatest cache hit ratio
- Deschedule if cache hit ratio is less than 65

The above ruleset will refuse to schedule if there is poor cache performance on the node (less than 79) and direct workloads to the node with the lowest hit ratio. Additionally, if the cache ratio falls to extremely low levels, there is disruptive contention on the node and TAS will try to deschedule it and place it elsewhere.

2.2.2 Platform Power Orchestration

Platform power usage can be managed at the orchestration level with TAS by focusing workloads away from nodes with high power usage in order to ensure headroom is preserved for workloads at runtime. This enables higher predictability in workload performance by preventing hard power limits from being hit. This is desirable in all workloads, but particularly in Cloud Network Functions, where predictable packet processing is of principal importance.

Additionally, TAS can be used to spread power usage across a greater or smaller number of nodes or to set a benchmark for node performance, which can be maintained as the number of workloads scales.

A TAS rule like the following would prevent workloads from being scheduled on nodes with high power usage:

- Don't schedule if power usage > 85 percent of thermal design power

2.2.3 Vision Processing at the edge

TAS allows intelligent scheduling of workloads, based on custom resource rules for devices, such as Vision Processing Units (VPU). This allows granular and customized control of their use across a data center, and schedules workloads to nodes with important underutilized resources.

Tight control of resource utilization is very desirable in edge applications where there are real limits on availability. VPUs, like the Intel® Movidius™ Myriad™ X VPU, are involved in edge AI image processing workloads for use cases like traffic monitoring, video security systems and facial recognition.

In a case where there are multiple possible edge nodes with image recognition available, TAS can ensure that a new VPU-intensive workload is scheduled on a node where there is more VPU capacity available.

A TAS rule like the following works to make this happen without need for further configuration:

- Schedule on node with lowest VPU utilization

2.2.4 Platform Resilience

Health indicators from the platform can inform placement decisions to ensure nodes with lower reliability, as measured using a health metric, are not candidates for critical workload placement. Proactive replacement of workloads can be done, ensuring workloads are moved to a healthy platform once there is an indication of faults on their host. This can be implemented using a composite metric that is the result of an external analytics process. Telemetry scheduling rules like the example below can increase overall workload resilience:

- Don't schedule if health metric is equal to 1
- Deschedule if health metric is equal to 2

In the above policy, `health_metric` is the result of a weighted algorithm considering metrics from multiple sources including Intel® Resource Director Technology (Intel® RDT), processor Performance Monitoring Units (PMUs), and Intel® Xeon® Reliability, Availability, and Serviceability (RAS) metrics. A value of 1 means that there is some significant issue on the system that should block deployments of critical workloads. A value of 2 means that there is a serious node fault and critical workloads should be descheduled and replaced on some other node.

These platform metrics are provided by the platform telemetry agent; collectd and its plugins are available from the OPNFV barometer project (refer to Barometer Home in Table 2). The platform metrics are provided by collectd for processing in Prometheus* and delivered as alerts to TAS.

To learn more about using TAS for workload resilience and service healing, refer to this White Paper.

2.3 Technology Description

TAS allows arbitrary, user defined rules to be put in place in order to impact scheduling in a K8s cluster. Leveraging the K8s descheduler, it can evict workloads that are breaking some rules in order to have it replaced on a more suitable node.

In a modern cloud computing cluster, there is a torrent of data that only certain subject matter experts know how to interpret and act upon. In scheduling workloads, operators know on which worker nodes a workload may perform better based on up to date utilization metrics. Likewise, certain telemetry values, or combinations of values, can be recognized as signs that a node has some serious problem that is interfering with workload operation.

The TAS policy system allows these insights to influence the scheduling and lifecycle placement process— turning implicit personal knowledge into formal, actionable information. As in the above use cases, the modified scheduling recommendations automate actions that would otherwise be done by operations technicians and increase resource utilization across the whole cluster.

2.4 Architecture

Figure 4 is a diagram of a three node Kubernetes cluster with a separate control plane node. The key components are:

1. **Telemetry Aware Scheduling:** the overall system which is made up of three interdependent components.
   a. **Telemetry Aware Scheduler Extender**: recommends nodes for pod placement based on metrics rules.
   b. **Telemetry Aware Policy Controller:** watches the state of critical metrics and marks violating nodes as such.
   c. **Kubernetes Descheduler:** picks up the pods due to be evicted, runs safety checks, and deschedules.
2. **Kubernetes Scheduler:** performs scheduling operations and looks for advice from **(1).**
3. **Metrics API:** supplies the metrics used by Telemetry Aware Scheduling.
4. **TAS Policy:** a set of rules made available as a policy resource in Kubernetes.
5. **Pod Spec:** specifically references the policy that should be used to influence the scheduling of this workload.
6. **Node collection agents**: collectd and node exporter.

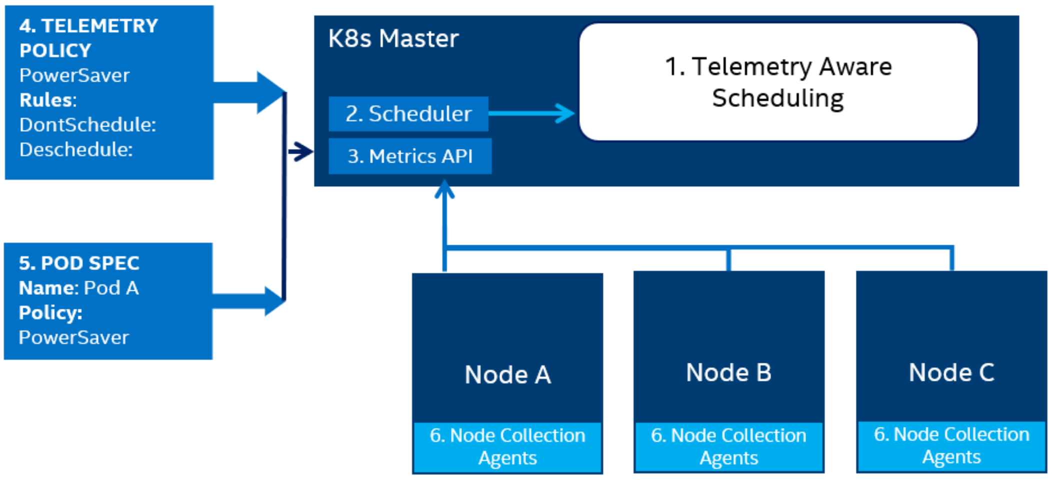

Figure 4. Architecture Diagram

3 TAS Software Deployment

TAS is an open source community project and the software is available at: https://github.com/intel/telemetry-aware-scheduling

Deployment is done directly on K8s*, using Helm* charts provided at the same repository. The following sub-sections describe the deployment process.

Note: These steps are based on TAS v0.1. Future releases may change aspects of the deployment process.

3.1 Overview of High Level Deployment Steps

The following steps are an overview of the dependencies and deployment process for TAS. It assumes that you are logged in to the K8s master node and you have the correct permissions to create and delete workloads and configurations.

1. **Install prerequisites**: Install programming languages and packages to deploy and build TAS.
2. **Install custom metrics pipeline**: Install the K8s metrics components required for TAS policies.
3. **Build, deploy and configure TAS**: Deploy the TAS system itself and configure K8s to run it.
4. **Deploy K8s descheduler**: Install the descheduler which enforces TAS deschedule policies.

A more detailed guide to deployment is available on the project repository.

3.2 Prerequisites

- Go* installation 1.12 or higher
- K8s installed on a multi-node cluster using Kubeadm/Kubespray
- Helm installed on the K8s cluster

3.3 Deploy the Prometheus* Custom Metrics Pipeline

The metrics solution is any pipeline that can supply metrics to the K8s Custom Metrics API. This example deploys Node Exporter, Prometheus, and Prometheus Adapter to provide metrics to the Metrics API.

The metrics pipeline contains three separate components:

1. **Prometheus**, the Cloud Native Computing Foundation monitoring project, is required in order to gather and store metrics for use in TAS.
   https://github.com/prometheus/prometheus
2. **Prometheus Node Exporter** is needed to collect platform metrics such as temperature and cache metrics, and make them available to Prometheus:
   https://github.com/prometheus/node_exporter
   CollectD is an additional complementary node metrics provider which makes many additional platform metrics available:
   https://github.com/collectd/collectd
3. **Prometheus Adapter**, which uses commonly used open source components to provide metrics, takes the metrics available in Prometheus, and makes them available through the K8s Custom Metrics API.
   https://github.com/DirectXMan12/k8s-prometheus-adapter

To deploy:

1. Clone the repository:
   ```
   git clone https://github.com/intel/telemetry-aware-scheduling
   ```
2. Move to the root of the cloned repository:
   ```
   cd telemetry-aware-scheduling
   ```
   There are several Helm charts in the repository that can be used to deploy a reference version of the metrics pipeline.
3. Install Prometheus and Node Exporter:
   ```
   kubectl create namespace monitoring
   helm install node-exporter deploy/charts/prometheus_node_exporter_helm_chart/
   helm install prometheus deploy/charts/prometheus_helm_chart/
   ```
4. Prometheus Adapter requires additional configuration for deployment, including the creation of certs on the host.
   To create self-signed certs for deployment of the adapter:
   ```
   openssl req -x509 -sha256 -new -nodes -days 365 -newkey rsa:2048 -keyout serving.key -out
   serving.crt -subj "/CN=ca"
   ```
5. Create the namespace and secret and then deploy the adapter:
   ```
   kubectl create namespace custom-metrics
   kubectl -n custom-metrics create secret tls cm-adapter-serving-certs --cert=serving.crt --
   key=serving.key
   helm install prometheus-adapter deploy/charts/prometheus_custom_metrics_helm_chart/
   ```
6. At this stage, the entire custom metrics pipeline should be installed and should be delivering node metrics from all nodes to the K8s API Server. To verify that this is working as intended:
   ```
   get --raw /apis/custom.metrics.k8s.io/v1beta1 | grep nodes
   ```

7. If this returns a list of entries, any single one can be viewed as a metrics object. These objects are returned as JavaScript* Object Notation (JSON*) and may be made more human-readable by using a program such as jq.
   To view a single metric entry for all nodes run:
   ```
   get --raw /apis/custom.metrics.k8s.io/v1beta1/nodes/*/load1
   ```
   Note: The above command returns metrics for all nodes in the cluster. If some nodes are missing, or no metrics are returned at all, there might be an issue at an earlier stage in the metrics pipeline.

3.4 Deploy and Configure TAS

Note: This guide assumes that the cluster was set up using Kubeadm or similar. The control plane components, especially the kube-scheduler, **must** be running inside the cluster as individual pods.

TAS is deployed on the K8s Master as a single pod with two containers. It requires a service and the cluster permissions to access resources such as metrics, policies, etc. in the K8s API server. Sample deployment files including Role Based Access Control permissions and service are available at the project repository in yaml format and in a Helm chart.

To configure a cluster for TAS deployment, you must make some changes to the configuration of the core K8s scheduler. These changes tell the system how and when to contact TAS for scheduling advice.

1. Deploy the scheduler config map:
   ```
   kubectl apply -f deploy/extender-configuration/scheduler-extender-configmap.yaml
   ```
2. Configure the scheduler so that it can locate this file and give it permissions to access configmaps by running:
   ```
   kubectl apply -f deploy/extender-configuration/configmap-getter.yaml
   ```
3. The config map can be accessed by the default scheduler by passing the following arguments to its binary, such as:
   ```
     - command:
       - kube-scheduler
   ```
4. Add:
   ```
       - --policy-configmap=scheduler-extender-policy
       - --policy-configmap-namespace=kube-system
   ```
5. All other arguments should be left unmodified, so the result is similar to:
   ```
     - command:
       - kube-scheduler
       - --policy-configmap=scheduler-extender-policy
       - --policy-configmap-namespace=kube-system
       - --bind-address=127.0.0.1
       - --kubeconfig=/etc/kubernetes/scheduler.conf
       - --leader-elect=true
       - --v=4
   ```
6. Additionally, in the same file, the dnsPolicy should be set to **`ClusterFirstWithHostNet`**. Directly above *volumes* in the manifest file, add the line `dnsPolicy: ClusterFirstWithHostNet`.
   The result will look similar to:
   ```
     dnsPolicy: ClusterFirstWithHostNet
     volumes:
     - hostPath:
         path: /etc/kubernetes/scheduler.conf
   ```

Now that the scheduler is properly configured, you can deploy the TAS system.

1. The first step is to create a secret that allows contact with TAS. This example uses the central cluster certs for verification.
   Note: In a production cluster, these certs should be set up as per normal cluster security policy.
   ```
   kubectl create secret tls extender-secret --cert /etc/kubernetes/pki/ca.crt --key
   /etc/kubernetes/pki/ca.key
   ```
2. Build TAS and deploy it on the cluster:
   ```
   make build && make image && kubectl apply -f deploy/
   ```
3. TAS should show it is running with the command:
   ```
   kubectl get pods -l app=tas
   ```

3.5 Deploy and Configure Kubernetes Descheduler

TAS relies on the K8S descheduler to remove workloads from a node. TAS communicates that a node is unsuitable for certain workloads using a label, and the descheduler can decide whether these workloads are safe to evict. If so, the TAS policy will direct the workloads to a more suitable node and the correct number of replicas will be maintained by the K8s system, while improving the quality (measured by the cluster operator) of the node on which the workload sits.

To install the descheduler on your cluster, run:
```
git clone https://github.com/kubernetes-sigs/descheduler
cd descheduler
make
cp ./_output/bin/descheduler /usr/bin
```

These commands install the binary to your path. For more information on configuring and running the K8s descheduler, refer to the repository at: https://github.com/kubernetes-sigs/descheduler

4 Implementation Example

After following the steps in Section 3, TAS should be up and running in the cluster. In order to use it, you must create a Telemetry Policy. This implementation uses a dummy metric (`health_metric`), which represents an analytics-based score of the health of a node. In this example, you will write the health metric to a file to prompt specific action from TAS.

After the deployment process in Section 3, the Prometheus* Node Exporter is set up to watch files in the folder `/tmp/node-metrics.`

To create a metric:

1. Run:
   ```
   echo “node_health_metric 1” >> /tmp/node-metrics/metrics.prom
   ```
2. For each remote node in the cluster, run:
   ```
   echo 'node_health_metric ' 0 | ssh <USER>@<NODE_IP> -T "cat > /tmp/node-metrics/metrics.prom"
   ```
   Note: You must enter the username and IP address of the node. You may also need to supply a password.
   This sets a health metric for each node in the cluster to **0** – or “**healthy**” under the terms set out in the use case in Section 3.
3. To see the metric in the custom metrics API, run:
   ```
   kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1/nodes/*/health_metric
   ```
   The result is some JSON* output showing the value of the health metric for each node in the cluster.
4. Deploy a policy that uses this health metric as an input. The following example policy is provided below and is available in the open source repository:
   ```
   apiVersion: telemetry.intel.com/v1alpha1
   kind: TASPolicy
   metadata:
     name: demo-policy
     namespace: default
   spec:
     strategies:
       scheduleonmetric:
         rules:
         - metricname: health_metric
           operator: LessThan
       dontschedule:
         rules:
         - metricname: health_metric
           operator: Equals
           target: 1
         - metricname: health_metric
           operator: Equals
           target: 2
       deschedule:
         rules:
         - metricname: health_metric
           operator: Equals
           target: 2
   ```
   This policy can be applied as-is with the command:
   ```
   kubectl apply -f deploy/health-metric-demo/health-policy.yaml
   ```
   The policy, named `demo-policy`, contains three separate rulesets.
   - The first tells the scheduler to direct workloads toward nodes with the lowest (that is, most healthy) `health_metric` reading.
   - The second “`dontschedule`” tells the scheduler to not even consider nodes with readings of 1 or 2.
   - The third informs TAS that this node is not suitable for running workloads and should be removed and rescheduled.

   If you schedule a pod that calls for TAS, you will be able to see the impact in the scheduler logs and in the scheduler result.

1. Run:
   ```
   kubectl apply -f /deploy/health-metric-demo/demo-pod.yaml
   ```
   Nodes with `health_metric` of 1 or 2 will be filtered out, while those of 0 will be passed on as possible candidates. In this case, there will be no extra prioritization done by TAS because all possible candidates have the same `health_metric` of **0**.
2. The decision made by TAS can be seen in logs using:
   ```
   kubectl logs -l app=tas -c tasext
   ```

3. Now that the pod is deployed, you can see the node it ended up on using:
   ```
   kubectl get pods -lapp=demo -owide
   ```
   After the pod has been deployed, its deschedule metric (which is the same as its scheduling metrics in this case) is being watched by the TAS Policy Controller for all nodes in the cluster. You can see this process in the controller logs by running:
   ```
   kubectl logs -l app=tas -c tascont
   ```
4. To see the descheduling strategy in action, find out what node the pod is currently running on and run the health metric script, setting the health metric, the deschedule target, in our policy. To do this in one command, enter the following command:
   ```
   echo 'node_health_metric ' 2 | ssh $(kubectl get pods -lapp=demo -o
   jsonpath='{.items[0].spec.nodeName}') -T "cat > /node-metrics/metrics.prom"
   ```
5. Now that the node has been inflicted with a critical error, as defined in the policy definition above, you can see the controller take note by looking at and following the logs:
   ```
   kubectl logs -f -l app=tas -c tascont
   ```
   You should be able to see the nodes being marked as violators of the demo policy.
6. The final step of the process is to run the descheduler in order to stop the workloads on the offending nodes. Run:
   ```
   descheduler --logtostderr -v4 --kubeconfig=$HOME/.kube/config --policy-config-
   file=$HOME/telemetry-aware-scheduling/deploy/health-metric-demo/descheduler-policy.yaml
   ```
   Note: The command above makes assumptions about the locations of the TAS directory and the kube config file. Your paths may be different.
7. The output of this process should show a pod being descheduled. Checking the TAS extender logs allows you to see the pod being rescheduled with the offending node now filtered out of the list of candidate nodes.
   ```
   kubectl logs -f -l app=tas -c tasext
   ```

5 Summary

This document offers a high-level overview of Telemetry Aware Scheduling on a standard K8s* cluster, describes the architecture model, and discusses concrete example use cases for TAS.

The implementation in this document allows domain knowledge of cluster health and workload performance metrics to be distilled into a clear, rule-based scheduling policy. The result can increase automation, improve resource management, and heighten fault tolerance in a K8s cluster. At a higher level, the improved scheduling system offers opportunities for lower operational expenses through systematic automation, and lower capital expenditure through improved platform utilization.

For more information on the benefits of TAS, including deployment scripts and instructions, refer to the open source repository at: https://github.com/intel/telemetry-aware-scheduling

0420/DN/PTI/PDF 621831-001US